DOCUMENT RESUME

ED 296 702 IR 013 328

AUTHOR Perkins, D. N.; And Others
TITLE Nontrivial Pursuit: The Hidden Complexity of Elementary Logo Programming. Technical Report.
INSTITUTION Educational Technology Center, Cambridge, MA.
SPONS AGENCY Office of Educational Research and Improvement (ED), Washington, DC.
REPORT NO ETC-TR86-7
PUB DATE Aug 86
CONTRACT 400-83-0041
NOTE 29p.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Cognitive Processes; *Difficulty Level; Elementary Education; *Problem Solving; *Programing; Psychological Studies
IDENTIFIERS *LOGO Programing Language

ABSTRACT

The thinking processes of students of Logo were examined to identify programming problems and possible instructional remedies. Subjects were 11 students between the ages of 8 and 12 who had completed 5 weeks of Logo instruction. These students were given a series of five short programming problems highlighting such areas of difficulty as judging angles, deciding on the directions of turns, using a variable, and using a subprocedure. The data collected included notes taken by the experimenter recording program errors, attempted repairs, and code written by students. A coding system was used to provide a measure of students' successes and errors in terms of the number of elements in a program they programmed correctly and their problem-solving efforts. The success rate in terms of elements correct was high, but success in terms of programs running successfully was lower, and a number of problems with what might be considered trivial aspects of Logo were recorded. A few students evinced serious problems with understanding tasks involving variables and a subprocedure. Possible explanations for the challenge of trivial elements of programming include: (1) the conjunctivity effect of minor problems; (2) a shortfall in elementary problem-solving strategies; (3) difficulty in discriminating concepts with superficial similarity; and (4) domain and domain operation problems. It is concluded that many trivial elements of Logo pose genuine conceptual difficulties, a problem that instruction must face and resolve. (25 references) (MES)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

86-7

ED296702

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

NONTRIVIAL PURSUIT:

THE HIDDEN COMPLEXITY OF ELEMENTARY LOGO PROGRAMMING

Technical Report

AUGUST 1986

Educational Technology Center
Harvard Graduate School of Education
337 Gutman Library Appian Way Cambridge MA02138

IR013338

BEST COPY AVAILABLE

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Beth Wilson

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Nontrivial Pursuit:
The Hidden Complexity of Elementary Logo Programming

Technical Report
August 1986

Writers:
D. N. Perkins
Michael Faraday
Chris Hancock
Renee Hobbs
Rebecca Simmons
Tara Tuck
Evelina Villa

Programming Group Members:

Betty Bjork
Susan Cohen
Michael Faraday
Chris Hancock
Michelle Harlow
Doug McGlathery
David N. Perkins
Rebecca Simmons
Steve Schwartz
Tara Tuck
Evelina Villa
Martha S. Wiske

Preparation of this report was supported in part by the Office of Educational Research and Improvement (Contract # OERI 400-83-0041). Opinions expressed herein are not necessarily shared by OERI and do not represent Office policy.

BEST COPY AVAILABLE

ABSTRACT

The authors observed step by step how 11 young students of Logo handled five very elementary Logo programming problems. While the students overall eliminated many bugs and achieved code about 90 percent correct, they only achieved correctly functioning programs only about two-thirds of the time, since even one mistake can spoil a program. Implications of this gap between "hit rate" for lines of code and correct programs are discussed. Genuine difficulties with understanding variables and subprocedures emerged in a few students, but overall impaired the students' performance no more than a number of seemingly more trivial mishaps concerning "mechanics"--syntax, sizes of turns, and the like. We conclude that many "trivial" elements of Logo pose genuine conceptual difficulties, a problem instruction must face and resolve.

Nontrivial Pursuit: The Hidden Complexity of Elementary Logo Programming

Recent research on young students of programming has shown that, very often, they achieve only modest competence (e.g. Dalbey & Linn, 1985; Kurland, Pea, Clement, & Mawby, 1986; Linn, 1985; Pea & Kurland, 1984a, b). Most writings on young students' difficulties with learning to program have highlighted the major conceptual barriers students face. Recursion, for example, is a puzzling programming concept even to most adults. Considerable experience with Logo may not suffice to give youngsters a sharp sense of the distinction between recursion and iteration (cf. Pea & Kurland, 1984a). Variables are another often-noted stumbling block. Nachmias, Mioduser, and Chen (1986) report a study of youngsters from middle primary school learning BASIC in which the younger participants encountered serious conceptual problems in mastering the use of variables, while succeeding well enough with more elementary aspects of BASIC. Even college students commonly evince some misunderstanding of the meaning of variables (e.g. Clement, Lochhead, & Monk, 1981), although experience with programming may actually help students' grasp of how algebraic expressions involving variables work (Ehrlich, Soloway, & Abott, 1982).

The use of subprocedures and structured programming is also an obstacle. Students tend to write "spaghetti code," for instance programming a complex graphics figure in Logo with a lengthy program rather than organizing it into subprocedures. According to Kurland, Clement, Mawby & Pea (in press), this reflects inclination as much as ability. While some students when pressed could not proceed in a more hierarchical fashion, others proved able to do so. Apparently, though, they did not normally feel moved to do so. Kurland, Clement, Mawby and Pea note that, from the students' standpoint, spaghetti programming is a straightforward strategy that minimizes cognitive load. Why strain?

Recursion, variables, and subprocedures are unquestionably important; they may even represent fundamental barriers to younger children's success with programming (Nachmias, Mioduser, & Chen, 1986). But why do they present the conceptual challenges they appear to? Presumably, complexity and abstraction in several senses figure in their difficulty. For one sense of complexity, some commands in certain languages have quite an intricate format -- the FOR-NEXT structure in BASIC, for example. However, by and large, the command formats in LOGO are quite simple. For another sense of complexity, some program elements afford a new order of complexity by participating in larger patterns of code. For instance, variables introduce a whole range of ways of shuffling data unavailable without them; subprocedures permit a complex hierarchical organization of code.

A third sense of complexity occurs when expressions can be imbedded within a particular program element. For instance, a REPEAT statement in LOGO can take an arbitrary expression evaluating to an integer as its first argument and any executable expression as its second. For another example, a subprocedure in LOGO can be defined with any particular number of arguments, and a given function call can include arbitrarily complex

expressions to be evaluated as the arguments before executing the procedure. There is reason to believe that complexity in general may impair learning through a cognitive load bottleneck as well as perhaps in other respects (cf. Brainerd, 1983; Case, 1984, 1985; Nachmias, Mioduser, & Chen, 1986).

Abstraction also is a likely contributor to difficult elements of programming. At least two senses of abstraction invite consideration. First of all, some program elements have an enormous range of potential reference, far broader than the "natural kinds" in terms of which we usually think. For instance a variable in LOGO can stand for any number, string, or indeed imbedded list structure. A second sense of abstraction is that the learner may lack a mental model or "envisionment" of the program element in question that helps to concretize it (cf. DuBoulay, O'Shea, & Monk, 1981; Gentner & Stevens, 1983). For instance, thinking of variables as boxes that have names and hold numbers or other data structures may be a useful envisionment of variables that some students lack. There is some reason to think that such mental models abet programming mastery (Mayer, 1976, 1981). Note that such mental models do not restrict the range of potential reference, the first sense of abstraction noted, but they nonetheless provide a more perceptual way of thinking about the program element. Note also that, in this second respect, how much of a problem of abstraction there is depends on what mental models the learner receives or constructs to concretize the programming element in question.

However, fundamental conceptual barriers are not the only source of difficulty that troubles novices. There is reason to think that young programmers often suffer from what we have previously termed a "fragile" knowledge base (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986). They not only have gaps in their knowledge concerning very elementary aspects of the programming language, but also possess a knowledge base problematic in other ways. Sometimes knowledge tends to be inert -- for instance, command knowledge is not retrieved on appropriate occasions but, on probing, proves to be present. On the other hand, sometimes knowledge proves too active -- commands and command elements migrate to places where they do not belong. For instance, in BASIC the STEP subcommands from a FOR-NEXT loop may be used in the midst of a PRINT statement. Such difficulties go beyond a simply spotty knowledge base. To describe the broader character of many novices' troubles with their knowledge base in programming, we have introduced the term fragile knowledge, which encompasses both knowledge gaps and other mishaps such as those described.

The fragile knowledge base commonly possessed by students is significant in several ways. For one, it warns that programming instruction cannot take for granted aspects of programming knowledge usually considered entirely elementary; they may bother students a good deal. It is not that students are less able than one might hope, but that the elementary features of programming are actually more complex than they seem from an adult perspective, a point that must be recognized in organizing appropriate instruction. For another concern, minor matters can do unusual mischief in a task like programming where everything must be correct for success; in contrast, a spelling mistake on an essay impairs the overall effect very little. On the positive side, there is some encouragement in the point that students'

knowledge is fragile, not just spotty. Students know more than they seem to at first, albeit in a fragile way, a point on which instruction might capitalize.

With the problem of fragile knowledge on our mind from prior analyses of BASIC programming (Perkins & Martin, 1986; Perkins, Martin, & Farady, 1986), we undertook a clinical study investigating how and how well young students of Logo handled a number of very elementary Logo problems. While looking for difficulties with variables and the use of a subprocedure, we also attended to difficulties in managing what might be called the routine mechanics of Logo. The goal of the study was not so much to test a specific hypothesis as to examine the thinking of students of Logo, to see what problems of programming gave students difficulty and to ponder what instructional remedies might help. The results lead us to reconceive in the discussion the ways in which elements of programming are trivial or not so trivial.

Method

Subjects

The subjects were eleven students who had completed a five week instructional sequence in Logo. The same five problems (to be described later) were to be given in the same order to each subject; all eleven subjects worked problems 4 and 5, but, because two students were late to class, only nine worked problem 1-3. The ages of the subjects ranged from 8 to 12. There were six girls and five boys of diverse ethnic backgrounds, including three Asian, one Mideastern and three Black students. Some of the students had had immediate mode experience with Logo prior to entering the program, but no more than that.

The five weeks of instruction were offered by the experimenters as part of a summer activity program at a local school. The classes were held twice a week, for two hours in the morning and one and one half hours in the afternoon. Each student had a computer to work with. The instruction introduced the students to the most basic elements of Logo, including REPEAT, variables, and subprocedures. This instruction simply provided the occasion for the clinical inquiry reported here. The instruction was not designed to address specifically the issues explored in this paper. Neither does the present analysis address directly the strengths or shortcomings of the instruction, but rather seeks to probe the nature of the students' understanding of Logo at the end of the course. We assume that the instruction was at least as competent as that received in most settings of Logo instruction -- not a strong presumption (cf. Delclos, Littlefield, & Bransford, 1985; Kinzer, Littlefield, Delclos, & Bransford, 1985; Pea & Kurland, 1984; Kurland, Clement, Mawby & Pea, 1986).

Programming tasks

In the experimental sessions at the end of the course, the investigators asked each student to attempt a series of five programming problems. Each problem involved writing a program to do some turtle graphics, illustrated on the problem sheet. Figure 1 shows the figures given to students to define the problems. To avoid issues of scale, the problem sheets for problems 1 through 4 indicated a segment length of 30 turtle steps (of course, the top segment of Problem 3 varied, as discussed below). The experimenters provided a subroutine for Problem 5 that settled the matter of scale.

Insert Figure 1 about here

The problems were very short, each designed to highlight a particular area of difficulty and not to pose other dilemmas. The areas of difficulty chosen for emphasis included the following: judging angles, deciding on the direction of turns, using a variable, and using a subprocedure. The choice did not reflect any theory of what was "really" trivial or nontrivial. Rather, the choice was motivated by our informal observations that students seemed to be having trouble with the matters mentioned, coupled with the received view in the field that subprocedures and variables pose problems for students. So we prepared five "diagnostic tasks" that would probe more formally whether students indeed were having difficulties in these areas and just what the difficulties were.

While only problems 3-5 involved traditionally "nontrivial" elements of programming, one should not think of problems 1-2 as the trivial problems and 3-5 as the nontrivial problems. First of all, what is really nontrivial and in what sense is at issue. Second, even traditionally challenging matters like the use of variables involve many routine aspects, such as including the variable name in a properly formatted program header. With these general points in mind, we comment briefly on each problem in turn.

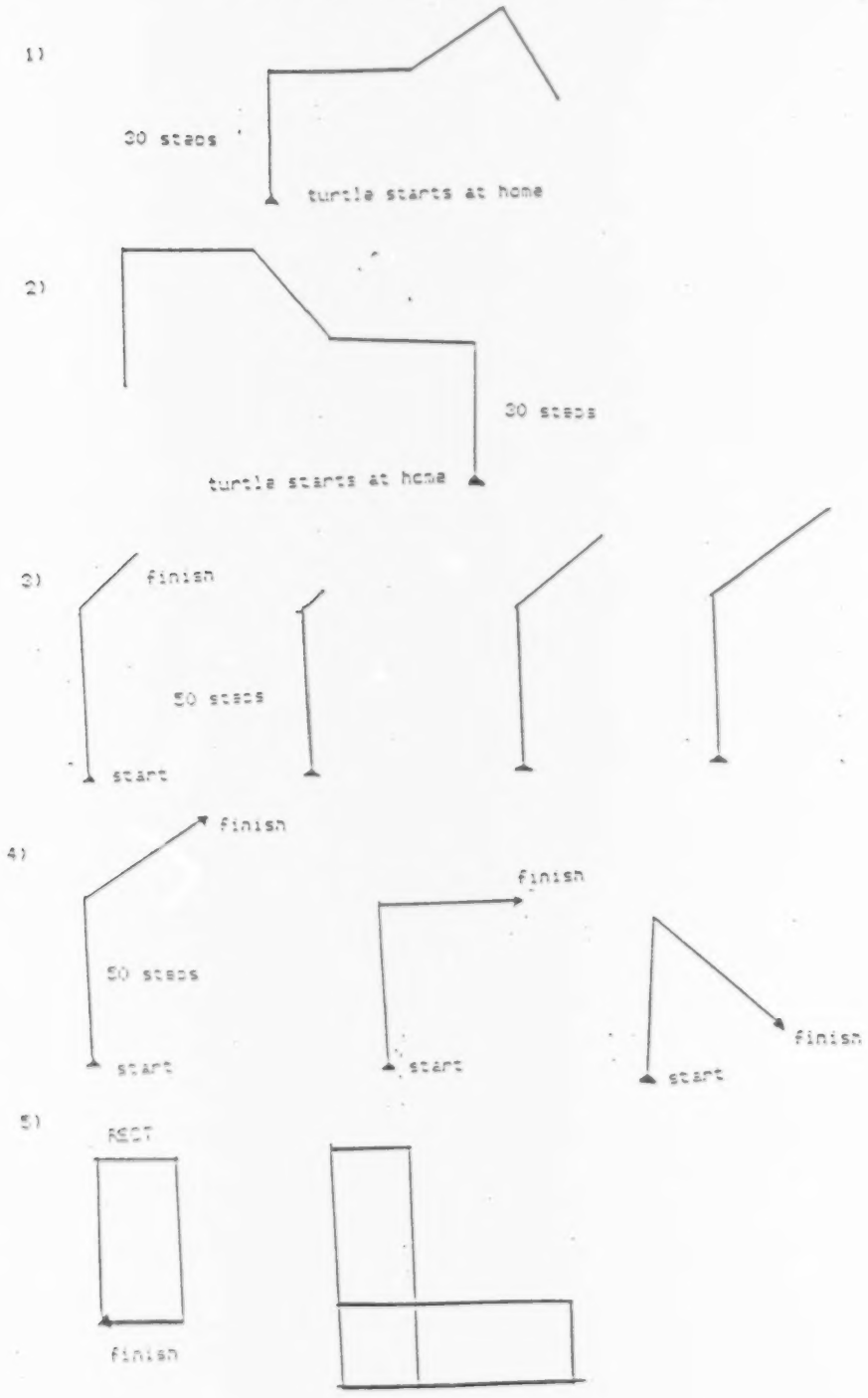
Problem 1. This problem focused on the turtle's orientation, incorporating a right angle tilted at 45 degrees that we suspected might give students difficulty.

Problem 2. This problem again focused on orientation, asking the students for a turtle path that navigated through two 90 degree and two 45 degree turns. We thought that the complexity of the path might present difficulties.

Problem 3. This problem required the use of a variable for length. The students were asked to write a single program that could make all four of the illustrated figures, each with a different length for the upper segment.

Figure 1.

Turtle paths for the five problems



The figures did not indicate the exact length of the variable segment and students were not scored wrong for using whatever lengths they cared to.

Problem 4. This problem resembled the previous, except that the variable concerned the angle rather than the length of the upper segment. Again, the students were not scored wrong for using whatever angles they wished in testing their programs.

Problem 5. This problem called for the use of a subprocedure. The experimenters provided the listing of a procedure called RECT that drew a rectangle, along with a picture of its output. The students were challenged to create the L shape displayed in Figure 1, using RECT twice.

Procedure

The problems were administered individually to the subjects by four experimenters working in parallel. The students worked on either Apple IIc or Apple IIe microcomputers, the same ones used in the instruction. At the outset, the experimenters made clear to the participants that the purpose of the study was to understand what aspects of programming were easy or hard to learn. As a student worked through a problem, the experimenter watched and took notes. Help was limited by policy: The experimenter could attempt to clarify the task if the student appeared not to understand what he or she was supposed to do, but could not give the answer to the problem.

The experimenter followed a set procedure for each problem. First, the experimenter explained the task to the student and asked the student to write a program away from the computer. Even though some children wanted to go directly to the computer, the experimenter insisted on a written program first. Then the student was allowed to go to the computer to try out the procedure and debug any problems that appeared. Some students chose to try out their procedures in the immediate mode first, although the students were encouraged to enter them directly as programs. If a student used the immediate mode, the student was then required to enter the procedure as a program and run it successfully before counting the student as having completed the problem.

The students had ten minutes in all to write a procedure, try it out, and debug it. If a student did not finish within this period, the experimenter pressed on to the next problem. Although ten minutes may not seem like much time, it should be remembered that the procedures called for were very short and were designed to present only one principal difficulty. Students with good mastery of Logo generally finished the problems in three or four minutes, even when needing to make a minor correction. Students having difficulty with a problem had sufficient time to try several repairs.

Data

Data collected during each session included notes taken by the experimenter, recording each mistake and attempted repair, as well as the code written by the students on their work sheets. All this was later combined by the experimenter into a protocol for each student for each problem. The scoring of the students' performances was based on these protocols.

Coding system

The experimenters developed a coding system with two objectives in mind: (a) to provide a measure of students' successes and errors in terms of how many elements of a program they got correct, rather than in terms of whether the program overall was correct; (b) to provide a broad "trace" of the students' problem-solving efforts. In particular, the system recorded the students' correct response or error and sequence of repair efforts for each element of code of each problem. Functions and arguments were scored separately. If an element was initially correct, it received a +, if initially mistaken, a - ("initially" refers by scoring convention to the version of the program written on paper, prior to entry into the computer). Each time that element was modified during the session, another + or - was appended to the scoring string, depending on whether the modification was correct or erroneous.

For example, suppose a RT 90 was initially rendered as a RT 60. The student received a + for the RT and a - for the 60. Now suppose that the student modified the RT 60 to RT 80. Nothing was added to the scoring string for the RT, because the RT had not been modified. But the scoring string for the 60 was changed to --, standing for the initial 60 and then the 80. Now suppose that the student corrected the RT 80 to RT 90. Again, nothing was added to the string for RT, but the string for the initial 60 now became --+, the + indicating a successful resolution.

This scoring system of course required that the scorers compare a line of code in a protocol to the "ideal" line of code. For this purpose, standard versions of each program were employed. The programs, recall, were very simple and the students virtually always approached them in essentially the same way. Even so, in principle ambiguities were possible. For instance, suppose a student wrote RT 30 when, in that area of the program a RT 90 and a FD 30 were called for. Would this count as a mistaken function and correct argument or a correct function and mistaken argument? To deal with such circumstances, several scoring conventions were established, as follows. It should be emphasized that the more specialized conventions rarely had to be applied.

1) What to score. The general rule was "one token, one score." For instance, FD 45 would receive two score strings, one for FD, one for 45. Revisions to the procedure were scored even if done in the immediate mode. Screen repairs -- for instance, efforts to fix on-screen errors by shifting

the pen to background color and erasing lines -- were not scored. Unnecessary and erroneous PU's and PD's were not scored. The only part of a program header that was scored was the input variable, if one was required. This was scored in the handwritten plan only if the student had actually written a header as part of the plan. Of course, many students did not write headers until they entered their programs as procedures.

2) Line matching. When there was a choice of what line of the model to match a line of code to, the match was made which would maximize the student's + scores. If this principle did not decide the issue, the earlier line in preference to the later one was matched. If the above two principles did not decide the issue, the function in preference to the argument was matched. If the student used a group of lines where he or she should have used a procedure call, the entire group was matched to the procedure call, the student receiving a - for it.

3) Good enough. If the student typed FD 28 instead of FD 30, or FD 89 instead of FD 90, this was scored as a +. In particular, distances and angles were counted as correct if no more than 3 units off. If a student typed FD 14 FD 14 FD 2 instead of FD 30, this was also considered as correct and scored as a single line. However, successive lines with the same function were not necessarily added for scoring; sometimes it made more sense to think of each as separate.

4) Skips, transpositions, and interpolations. A skipped line was counted as a double error, - for the function and - for the argument. If the student's version of a program did not require a line that was in the standard program, it was, of course, not scored as omitted. A pair of transposed lines was indicated with a single minus at the bottom of the scoring page, apart from the normal line sequence and labelled with the line number. If a transposition did not appear at the outset, but only later, it was marked +- to show the initial correctness. Besides this, transposed lines were each scored normally for the correctness of their content. Unnecessary interpolated lines were marked at the bottom of the page also. These were rare and were not included in the final tallies discussed below.

Scoring procedure

Using the scoring system, three scorers independently coded all the protocols. They checked periodically with one another after scoring a few protocols to work toward adequate calibration, establishing some of the rules above as they went. Policies that could affect prior scoring were applied retroactively. Although disagreements in scoring were discussed and resolved, the original codings, prior to discussion, were preserved for calculation of interjudge agreement later. After about half the data had been coded, the scorers deemed themselves to be adequately calibrated, although crosschecking continued to the end. The remainder of the coding was treated as the basis for calculating interjudge agreement, while all the coding was used for the data analysis in other respects, on the grounds

that divergences had been discussed and resolved and principles applied retroactively to the coding before the calibration point.

Tallying

The coded protocols were condensed further into a set of tallies for each subject-problem. One set of tallies was made for function problems, another for argument problems. For functions and arguments separately, for each subject and problem, the following counts were made:

Initially correct (+). Single +'s represented initially correct and never modified program elements.

Immediately corrected (-+). These strings corresponded to program elements initially incorrect, but corrected in a single trial.

Eventually corrected (...+). These were program elements incorrect initially or perhaps later changed to something incorrect, but ultimately corrected. All involved at least two modifications.

Unmodified errors (-). These were program elements initially incorrect and never modified.

Modified errors (...-). These were program elements initially or later incorrect and, although changed, never properly corrected.

Results

Interjudge agreement

Interjudge correlations were calculated to test the reliability of the scoring procedure. In the functions category, correlations ranged from 1.0 (for several tallies) to .891 ($p<.001$, $N=24$) for eventually corrected elements. The mean correlation was .966. The correlations for the arguments category ranged from 1.0 (for several tallies) to .688 ($p<.001$, $N=24$) for unmodified errors. For arguments the mean correlation was .915.

Profiles of performance

Table 1 provides a profile of performance, pooled over students, separating functions and arguments. The subtotal rows show that arguments generally posed more of a problem than functions. The total figures reveal that the students as a whole did fairly well as gauged by program elements. After debugging, only 8 percent errors remained for a percentage correct of 92. However, their success rate as measured by percent of programs running successfully was much lower, as one would expect: 67 percent.

Insert Table 1 about here

The figures also argue that the students in general did not become mired in difficulties. A natural conjecture would be that students who could not correct a difficulty immediately might make little further progress on it. To the contrary, the figures for "immediately corrected" and "eventually corrected" show that the students often continued to make progress even after an initial effort at repair had failed. The modest number of "unmodified errors" indicates that students made attempts to repair nearly all their errors, although not always successful ones.

Table 2 provides a profile of the students' performance student by student, offering for each problem number of initial errors and number of final errors, pooled over functions and arguments. As one would expect, the table reveals a considerable difference in performance across the students. Student 1, for example, made many initial errors and sometimes proved able to correct some. Student 2 made far fewer initial errors but evinced considerable difficulty in making corrections. Student 6 made very few initial errors. These profiles reflect our experience with the students. A couple were "whizzes," doing the problems virtually effortlessly, while, at the opposite extreme, another couple displayed considerable confusion.

Insert Table 2 about here

With this general picture in mind, we turn to a problem by problem interpretive discussion of the difficulties students displayed.

Problem 1

The tables show this problem to be a reasonably tractable one. By the end of the problem period, only 1 error out of 63 total functions remained, and 7 errors out of 63 total arguments. The students, without exception, wrote the first three lines of the program (FD 30, RT 90, FD 30) without error. However, as soon as the target image no longer followed the lines of a normally oriented square, the difficulties began. For the second turn of the program (LT 45), five out of nine students incorrectly coded the angle in their plan. Three of the students coded an incorrect direction (RT instead of LT) and two coded an incorrect angle.

The third angle, a right angle oriented askew to the horizontal -

Table 1

Profile of Performance:

Functions and Arguments Tallies

Problem	Initially Correct	Immediately Corrected	Eventually Corrected	Unmodified Errors	Modified Errors
		Function Tallies			
1	57	4	1	0	1
2	70	5	0	4	3
3	27	0	0	0	0
4	33	0	0	0	0
5	31	3	7	0	9
subtotal	218	12	8	4	12
		Argument Tallies			
1	52	4	0	4	3
2	72	2	0	2	4
3	21	7	3	0	5
4	34	0	3	0	7
5	10	5	7	0	6
subtotal	189	18	13	6	25
total	407	31	21	10	37

Table 2

Student Performance:

Initial Errors / Final Errors

Student	Problem				
	1	2	3	4	5
1	5 / 4	8 / 7	3 / 3	2 / 2	7 / 0
2	1 / 1	1 / 1	2 / 2	2 / 0	3 / 3
3	2 / 0	2 / 0	2 / 0	1 / 0	5 / 0
4	1 / 0	2 / 0	2 / 0	0 / 0	1 / 0
5	0 / 0	0 / 0	3 / 0	2 / 2	2 / 2
6	3 / 0	0 / 0	0 / 0	0 / 0	0 / 0
7	3 / 2	0 / 0	1 / 0	0 / 0	1 / 0
8	2 / 1	5 / 5	2 / 0	0 / 0	4 / 4
9	0 / 0	2 / 0	0 / 0	0 / 0	3 / 0
10				0 / 0	5 / 0
11				3 / 3	8 / 8

vertical frame, posed a difficulty of argument out not function. While six of the nine students incorrectly coded this turn, all six coded the direction correctly; it was the angle that led to error. One student's comments illustrated the nature of the difficulty especially well. Laurie had originally coded the third angle as a RT 60. When she saw the picture on the screen, she localized the problem with no difficulty, but did not know what the number of degrees should be, stating, "Oh, I think I need 70 maybe." She began a series of steps in immediate mode to correct the error, but this led into more difficulties. When the experimenter asked her what she was attempting to do, she began to explain, then interrupted her own sentence with the exclamation, "Right 90!" The experimenter asked her why she felt it was a right 90 turn. Her response: "I looked at it this way [she turned her head], and it had a 90 degree angle."

While most of the students easily identified problems of a mistaken angle (either the 45 or the 90 degree angle), finding the correct angle was more difficult. These students did not seem to have learned the look of the two angles, even though the instruction had stressed the usefulness of both. In addition, two of the students attempted corrections in ways that showed a misunderstanding of the nature of a unit change in degrees. For example, one student had written RT 29 instead of RT 90. She had no difficulty in deciding that the angle needed alteration. However, her successive corrections were 34, 27, and finally 32 degrees, at which point her time ran out. Apparently she had little notion of the magnitude of a degree.

Problem 2

Tables 1 and 2 show that students managed Problem 2 fairly well, with 7 errors out of 82 function elements and 6 errors out of 80 argument elements remaining after debugging. Besides posing more 45 and 90 degree angles, the problem required the students to trace a path proceeding leftward rather than rightward. The tendency of the instruction had been to employ rightward turning paths, perhaps establishing a kind of "template" of restricted lability. Three of the students incorrectly coded the first turn, a LT 90, two of them indicating the RT rather than the LT function. The third displayed an interesting twist. His first try at the turn was LT 30. He localized the error with no difficulty but substituted RT 90, changing both direction and degree. One could posit that, when in trouble, he reverted to the familiar rightgoing template.

In problem 2, students seemed to have less difficulty in determining 45 and 90 degree angles, although they still confused the two and displayed difficulties in choice of RT versus LT. In particular, in Problem 1, six students at some point estimated an angle as something other than 45 or 90 degrees. In the second problem, only two made such misestimates, both of them having done likewise in Problem 1. The greater attention to 45 and 90 degree angles might reflect experience with Problem 1.

It was encouraging to observe that, in Problems 1 and 2, a couple of the students used their mathematical knowledge of angles to motivate

corrections, in contrast to many students who seemed to tinker "from the picture." These latter students quickly modified incorrect angles, noted the changed picture on the screen, and from that determined whether a larger or smaller angle was needed, apparently without much reflection. In contrast, consider one student who had written a RT 45 instead of a RT 90. Rather than replacing it with a RT 90, she inserted another RT 45 in her program. When the experimenter asked her why she chose to add another 45 degrees she answered, "To get it another 45 down." In another instance, a student had written a LT 45. She looked at her picture and said, "Oh, right 45." The observer asked her how she knew it was 45 degrees, to which she responded, "Because I knew half of 90 was 45."

Problem 3

Problem 3 focussed entirely on introducing a variable and involved only three lines of code plus the program header. Not surprisingly, functions proved no challenge at all; 100 percent of students chose them correctly on their first trials. But the introduction of the variable posed problems, yielding a final error rate of 5 out of 36 on arguments. Only one student incorrectly estimated the RT 45 turn, even then correctly planning it but making an error later when he began to get tripped up in coding the variable. The distance variable posed many difficulties for the students. The problems ranged from students who did not even understand that the task required a variable to those who had a good grasp of the concept of the variable and recognized the need for one in the program but had trouble recalling the correct syntax.

For instance, consider a student who appeared not to understand the basic concept of a variable. The experimenter explained the problem to him, stressing that the program must be able to draw any of the four sample outputs provided (involving a variable line length). When the experimenter asked how he might do that, he responded, "Easy -- just go to the editor and type them all in." He wrote three lines: FD 30, RT 45, FD 20. When again reminded that the program must be able to produce any of the figures, the student went back to his planning sheet and added the following lines to his original program: FD 15, RT 45, FD 10, FD 50, RT 45, FD 30, FD 100, RT 45, FD 110. He then told the experimenter, "This is all four programs; I'll put it in the editor," and proceeded to do so.

Some students recognized that a variable was needed but had difficulties placing it in the program. One student called the variable SIDE and correctly inserted it in the header but then used it for the angle rather than the final distance. Another student originally coded the variable in both distance commands. A third student correctly determined that the variable should be used with the third command, involving distance. However, when she was planning the program she said to herself, "Is it distance or angle?...It's angle," and proceeded to call her variable ANG. Another common mistake was to forget to put the colon before the variable. Despite their difficulties, seven of the nine students produced entirely correct programs by the end of the time period.

Problem 4

Another variables task, Problem 4 showed a pattern of difficulties and successes resembling Problem 2. Again, all the students selected correct functions on their first trials. The final success rate on arguments fell out nearly the same as well -- 7 out of 44 arguments. The two youngest students proved unable even to grasp the problem. Despite repeated instructions that one program should be able to make all three shapes, with the experimenter breaking the rules a bit to show the students on paper how the program should be called to produce each of the three shapes, these two students insisted on writing separate sequences of Logo commands for each of the three example shapes shown. One other student came very close to solving the problem but never got her program working due to syntax errors that she could not surmount. The remainder of the students produced working programs with only minor difficulties, although one student had to be reminded of the syntax for her inputs. It appears from this that the hardest aspect in mastering input variables may concern understanding how they might be used rather than how to implement them. Were this not the case, we would expect to find more students at a middle stage -- able to grasp the problem but unable to solve it.

While Problem 3 called for a variable to represent length, Problem 4 demanded one to represent angle. We speculated that some students might manage Problem 3 but not Problem 4 because their understanding of inputs was too bound to the particular case of inputs for distance. This conjecture was not fulfilled. In fact, every student who solved problem 3 was also able to solve problem 4. However we were intrigued by the names chosen by the students for their input variables -- :a, :ang, :d (two students), :s, :side, :step, and :x. Two of these names (:side and :step) clearly refer to distance, and three more (the two :d's and the :s) may also refer to distance. For at least of some of the students, then, the understanding of inputs still appears somewhat attached to the original context of learning.

Problem 5

The only task to involve subroutines, Problem 5 also yielded the highest error rates -- 9 final errors out of 50 on functions and 6 out of 28 on arguments. What difficulties did students display? One student did not grasp the idea of using a procedure to make a chunk of a larger drawing. She insisted on writing a single long procedure to make the "L" shape. Two other students displayed an intermediate level of understanding. They formulated a sequence of immediate mode steps:

```
RECT    RT 90    FD 15    RT 90    RECT
```

that produced a fine "L", but they proved unable to incorporate these steps into a superprocedure. Instead they tried editing the RECT procedure to add more lines at the bottom (one of the two actually put a recursive RECT call at the very bottom, below the three repositioning commands). When we

insisted on a separate procedure for the "L," these students reproduced the contents of the RECT procedure in the new L procedure.

Evidently these students had learned the first important fact about using subprocedures: "A procedure can be used to help make a bigger picture." But they had not learned the second one: "A procedure can call another procedure." This is, of course, just a particular case of a more fundamental Logo principle: "You can put into a procedure any sequence of commands that you can execute in immediate mode." Clearly if our two students had understood this rule fully they would not have performed as they did. It might be argued that our class, and the students' previous Logo classes, did not stress this principle sufficiently. We favor the interpretation that the principle is too abstract to help most students over the hump of calling procedures within procedures.

The seven students who succeeded in solving the problem displayed an interesting variety in planning. One student who appeared to understand how to use inputs was distracted and did not complete the problem. Two students used the definition of RECT, which we provided, to help plan their superprocedures. They reasoned correctly from our code for RECT that the turtle would have to turn twice and that it would have to move 15 turtle steps between turns. Two others did not use such detailed information. They understood that RECT produced a rectangle but did not check closely to see the rectangle's dimensions or the turtle's final position and orientation. Both of these students missed the fact that the turtle would need to make an initial turn, and both at first estimated that the turtle's repositioning move should be 20 turtle steps. They later fixed their programs, but by trial-and-error in immediate mode, not by looking at the definition of RECT. Finally, two students planned without closely checking the definition of RECT, but returned to it when their first effort did not work, and were able to use it well. The case of these seven students shows a clear advantage for those students who can reason precisely about written procedures.

Discussion

This study aimed to develop a profile of the young LOGO students' programming difficulties, with particular attention to "trivial" problems that might turn out to be more significant than one would at first suppose. The combination of formal scoring and interpretive analysis of individual protocols provided a fairly sharp picture of the students' problems and successes. Besides addressing Logo specifically, we also suggest that the general conclusions apply to initial learning of other programming languages.

General effectiveness in programming and debugging

The tabulations of results showed that the students overall were rather effective in programming and debugging these elementary problems.

Pooling figures for functions and arguments, the students proved initially correct on 82 percent of the program elements (functions and arguments combined) and gained another 10 percent through debugging for a final 92 percent elements correct, a very respectable "score."

The students taken as a whole also revealed significant powers of working through a difficulty to a conclusion. The students were able to correct somewhat more than half of their initial errors. One might suppose that the students would have little chance of correcting a problem that was not just a "quick fix." However, the tabulations show that the students resolved about as many bugs through multiple trials as through a single modification. Although the students displayed very different degrees of success with the problems, all disclosed some ability to diagnose and correct difficulties.

The "conjunctive task" effect

Although the overall success rate measured in terms of elements correct was quite high, the students' success rate measured in terms of problems solved correctly was much lower. In particular, the students taken together produced correctly functioning programs 67 percent of the time. As a "hit rate," this is not nearly so respectable as the 92 percent correct for program elements after debugging.

The reason for the difference is transparent. Unless one is fortunate enough to have two mistakes cancel each other, a single erroneous element in a program prevents it from running correctly. Accordingly, one can have quite a high accuracy rate as measured by correct elements with a substantially lower success rate as measured by correct programs. This would hardly be worth mentioning were it not that many tasks characteristic of schooling do not have this property. Spelling tests, multiple choice quizzes, and essays normally are assessed on the basis of the preponderance of correct, or, in the case of an essay, sound, elements.

However, programming, arithmetic problems, and a few other formal tasks such as proofs in Euclidean geometry are what might be called "conjunctive" tasks. Conjunction is used here in the sense of the logical "and;" a correct response to a conjunctive task depends on the correctness of each individual response element. Any error spoils the whole. Accordingly, conjunctive tasks -- whether in programming or other subjects -- demand very high precision for good overall performance as measured by number of tasks correctly completed.

One might think at first that the usual way of assessing performance on conjunctive tasks is unfair. Imagine how much higher arithmetic scores would be if students were graded by the number of elemental operations correctly executed rather than the percentage of right answers! But the fact is that, for conjunctive tasks, in the end it <u>is</u> the whole task that counts. A program does not work unless it is virtually error-free; the answer to an arithmetic program does not serve unless it is correct. A flawed response to a conjunctive task is largely nonfunctional. A very

high level of precision is required for fruitful results, and this pressure for precision must be taken into account in comprehending students' difficulties and designing instruction.

"Trivial" problems

The interpretive analyses of the individual protocols recorded a number of problems with what might be considered trivial aspects of Logo. Mix-ups between LT and RT were not uncommon. The angle of 45 degrees was not always recognized. The angle of 90 degrees often was not recognized when it occurred in nonstandard orientation, and once, in normal orientation, was treated as 30 degrees. In addition, students working through such difficulties often proceeded in a trial and error fashion, twiddling parameters in ways that sometimes suggested little grasp of whether an angle needed to be bigger or smaller or how large a degree was.

While variables and subprocedures sometimes presented serious problems, "trivial" problems also appeared in association with them. Students recognizing the need for a variable often had difficulty recovering the correct format for the header and for using the variable in the program. While employing the subprocedure for a rectangle in a program, some students had difficulty positioning the turtle with intermediate moves for a successful call of the subprocedure.

How trivial, then, were these "trivial" problems? On the one hand, certainly many of the students had considerable success in resolving them. In keeping with the concept of "fragile knowledge" mentioned in the introduction, for the most part these were not matters on which the students had no hold at all. They usually could and did take steps to remedy the matter, revealing more knowledge than was apparent at first in their often dismayingly elementary mistakes. On the other hand, these trivial problems did not always go away when attended to. For instance, problems 1 and 2 involved no element of variables or subprocedures, yet only 61 percent of the programs were entirely correct after debugging efforts.

"Deep" problems

As noted in the introduction, mastery of variables and of use of subprocedures usually are considered serious conceptual challenges for young programmers (cf. Pea & Kurland, 1984; Kurland, Clement, Mawby & Pea, in press; Nachmias, Mioduser, & Chen, 1986,). Students often evince fragile knowledge here too -- the partial mastery, inert knowledge, and mistaken migrations of knowledge characteristic of the fragile knowledge syndrome. But a few of the students evinced serious problems of understanding on the tasks involving variables and a subprocedure. In particular, they appeared not to be able to approach the variables problems (3 and 4) by using variables, but instead persisted in writing out code for each separate case. On the subprocedure problem (5), one student insisted on spaghetti coding the program even though the subprocedure was provided; two others

could use the RECT subprocedure in immediate mode but appeared unable to incorporate it into a procedure.

On the other hand, it should not be thought that the net effect of these challenges was any greater than those of the "trivial" problems. This "programs ultimately correct" figure for the variables problems was 75 percent and for the subprocedure problem 64 percent, a bit larger than the 61 percent for problems 1 and 2, which did not involve these concepts. Moreover, some of the difficulties students experienced in the variables and subprocedures problems were "trivial" matters. To be sure, these problems were, designedly, very elementary applications of variables and subprocedures. Nonetheless, it is interesting to note that the students as a whole did not appear to find in them challenges any more troublesome than those posed by problems 1 and 2.

What makes an element of programming nontrivial?

The results of our study suggest that "trivial" elements of Logo programming are not so trivial as they might seem. Because of other research, we suggest that the same can be said for BASIC, PASCAL, and other programming languages (cf. Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Perkins & Martin, 1986; Sleeman, Putnam, Baxter & Kuspa, 1986). Perhaps there is a need in the psychology of programming to overhaul notions of what is trivial and what is not. In the introduction, we noted that complexity and abstraction in several senses helped to explain the difficulties posed by programming elements such as variables and subprocedures. These concepts might apply somehow to elements of programming usually considered trivial. Moreover, there might be other factors that make seemingly mechanical aspects of programming troublesome after all. We suggest at least the following explanations for the surprising challenge of "trivial" elements of programming.

The conjunctivity effect: Complexity in a new form. We have emphasized already that learners' knowledge of the mechanics of programming often is "fragile" -- not only spotty but often inert and sometimes used in garbled ways (Perkins & Martin, 1986). Taken piece by piece, much of fragile knowledge might be considered trivial; many a knowledge gap, element of inert knowledge, or garbled line of code would yield simply to "knowing the facts" better. Indeed, the students' fragile knowledge notwithstanding, many of them did quite well with the problems in our experiment.

However, the phenomena observed warn of much greater difficulties with more complex programs. The longer a program or the more subprocedures involved, the more amplified is the conjunctivity effect. Programs easily can become a tangle of errors exceedingly difficult for students to sort out. We note informally that the majority of the students at this point in the instruction had considerable trouble getting through a turtle graphics programming activity of modest scope -- say, a program to draw a tic-tac-toe design -- largely because of the erosive effects of minor problems. Recall that complexity appeared to be one source of students'

difficulty with challenging concepts like variables and subprocedures. We see here that, although many fragile knowledge problems in themselves are not particularly complex, complexity reenters the scene as a factor because of the conjunctivity effect.

Elementary problem-solving strategies. Coping with the conjunctivity effect can pose major cognitive challenges of task management. So can other aspects of fragile knowledge and, of course, programming in general. In research on novices' BASIC programming, we obtained results suggesting that many novices neglect and could use with profit quite elementary problem solving strategies to track goals, access inert knowledge, test candidate solutions, and deal with like matters (Perkins & Martin, 1986). For instance, students could ask themselves questions like, "What command do I know that does this sort of thing?" or "What does the line of code I just wrote really do?" While we did not investigate this question directly in the Logo study, we see much the same phenomena of fragile knowledge as in the BASIC study, so it is reasonable to presume that elementary problem solving strategies may be relevant to young Logo programmers as well.

Close discrimination problems. We already noted that complexity and abstraction of certain sorts may make elements of Logo and other programming languages difficult for learners. Another source of difficulty seems to be solution elements that are easily conflated and require careful discrimination. For instance, a number of students in the present study had problems differentiating left from right turns. It may seem odd to think of left and right as in some sense close to one another, but in psychological fact left-right discrimination tends to be troublesome in human perception: this is not just a matter of associating the appropriate names with left and right but of actual lability in the perceptual encoding of left versus right (cf. for instance Kolers & Perkins, 1969a, b). Some students also evinced difficulty in discriminating just where the turtle began in the RECT subroutine. Certain other troubles evinced by the subjects might be considered problems of discrimination as well.

Other familiar discrimination problems in Logo include differentiating between iteration and recursion (cf. Pea & Kurland, 1984a, b) and between reference and naming (:x versus "x for example). In other problem domains, problems of close discrimination are frequent. Whenever two concepts have a strong superficial similarity, students routinely and understandably evince problems disentangling them. Examples from physics include weight versus mass versus density and force versus pressure. The problem of close discrimination might be considered complexity in another guise; however, it is not just complexity in a loose general sense but a particular sort of complexity, where two conceptual structures look superficially alike, as in mass and weight. Therefore, problems of close discrimination seem worth singling out as a distinctive category of difficulty.

Domain operations versus control structures. The right-left confusions many subjects experienced point to another interesting generalization: Students evinced a number of problems that concerned geometry and the Logo commands for making geometric moves, such as LT and RT. Besides the left-right problem, some students failed to recognize a tilted right

angle. One appeared to have little sense of the magnitude of a degree. Several estimated angles poorly.

With such problems in mind, it is useful to distinguish roughly between the control structures and the domain operations of a programming language. We will consider the control structures of Logo to include subprocedures, recursion, REPEAT, and also variables; these are all tools to control program execution and manipulate general unspecified "objects." The domain operations of Logo as typically taught concern the turtle "object" and the world of turtle geometry: FD, BK, RT, LT, PU, PD, and so on. While the control structures are quite general, the domain operations are tuned to deal with a certain sort of domain. Of course, Logo is a general purpose programming language and can be used to deal with other "worlds" perfectly well. But the language is particularly tuned to turtle geometry. Adding other primitives or packages of subroutines could tune it to other "worlds" too. Analogously, the domain operations of BASIC tend to focus on the manipulation of arithmetic data; BASIC is tuned to the "world" of numbers and number arrays.

With this loose distinction in mind, we can make the following point. The conceptual difficulties in programming usually are presumed to lie in the general control structures, perhaps because they seem more complex and abstract at first thought. However, the domain operations and the domain itself may pose equally confounding problems of complexity, abstraction, and close discrimination. In the present study we see many subjects experiencing such problems with turtle geometry, and indeed one of the signal points about Logo has always been that it might teach students some geometry (Papert, 1980). Likewise, while FOR loops often pose problems for students of BASIC, so do more domain-oriented matters like arrays and the differentiation between strings of numerals and numbers, e.g. "123" versus 123.

Summary and implications for a better pedagogy of programming

Certain elements of programming are widely recognized to pose difficulties, presumably because of:

> Complexity in at least three senses: the element participating in more complex program structures, the element itself having a complex structure, and the element accommodating complex subexpressions.

> Abstraction in at least two senses: the element having very broad reference or application, and an absence of mental models that help one to visualize or otherwise make concrete one's understanding of the element.

However, the results of the present experiment, buttressed by analogous findings in other sources (e.g. Perkins & Martin, 1986; Perkins, Hancock,

Hobbs, Martin, & Simmons, 1986; Sleeman, Putnam, Baxter & Kuspa, 1986) argue that other aspects of programming that might be thought to be 'trivial' -- mere matters of 'knowing it' -- offer substantive challenges of their own. Without any pretense of exhaustiveness, we have identified the following concerns:

Close discrimination problems, which make certain concepts hard to grasp.

Domain and domain operation problems, which can be quite as troublesome for learners as problems having to do with the general control resources of the programming language.

The conjunctivity effect, which, in a high-precision endeavor like programming, escalates difficulties that in isolation may merely be matters of "knowing it" into a major challenge.

A shortfall in elementary problem-solving strategies, which prevents students from making the most of their somewhat fragile knowledge bases.

Such problems become apparent when, as in the present clinical study, one looks closely at difficulties students have that often are considered trivial -- mere matters of knowing left from right, remembering the syntax of a command, and so on. Accordingly, we suggest that a good pedagogy of programming depends on reconceiving what is hard and what is easy by recognizing the greater range of ways in which something can be hard. In particular, one cannot depend on broad formulas like "more practice" to repair students' difficulties with supposedly elementary aspects of programming, saving the more refined pedagogy of mental models and whatnot for the "truly difficult" concepts like the hierarchical use of subprocedures. Rather, there is a need to understand better the quite genuine difficulties posed by seemingly simple matters and to design pedagogy to deal with them. Such attention appears called for by this and other evidence urging that the many seemingly trivial problems of elementary programming add up to a distinctly nontrivial pursuit.

References

Brainerd, C. J. (1983). Working-memory systems and cognitive development. In C. J. Brainerd, (Ed.), Recent advances in cognitive-developmental theory: Progress in cognitive development research (pp. 167-236). New York: Springer-Verlag.

Case, R. (1984). The process of stage transition: A neo-Piagetian viewpoint. In R. J. Sternberg (Ed.), Mechanisms of cognitive development (pp. 19-44). New York: W. H. Freeman and Company.

Case, R. (1985). Intellectual development: Birth to adulthood. New York: Academic Press.

Clement, J., Lochhead, J. & Monk, G. (1981). Translation difficulties in learning mathematics. American Mathematical Monthly, vol 88 no.4, pp 286-290.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1, 253-274.

Delclos, V. R., Littlefield, J., & Bransford, J. D. (1985). Teaching thinking through Logo: The importance of method. Roeper Review, 7(3), 153-156.

DuBoulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. International Journal of Man-Machine Studies, 14, 237-249.

Gentner, D., & Stevens, A. L. (Eds.). (1983). Mental models. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Ehrlich, K., Soloway, E., & Abott, V. (1982). Transfer effects from programming to algebra word problems: A preliminary study (Report no. 257). New Haven: Yale University Department of Computer Science.

Kinzer, C., Littlefield, J., Delclos, V. R., & Bransford, J. D. (1985). Different Logo learning environments and mastery: Relationships between engagement and learning. Computers in the Schools, 2(2/3), 33-43.

Kolers, P. A. & Perkins, D. N. (1969a). Orientation of letters and errors in their recognition. Perception and Psychophysics, 5(5).

Kolers, P. A. & Perkins, D. N. (1969b). Orientation of letters and their speed of recognition. Perception and Psychophysics, 5(5).

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. New York: Bank Street College of Education, Center for Children and Technology. Also, Journal of Educational Computing Research, in press.

Kurland, D. M., Clement, C., Mawby, R., & Pea, R. D. in press. Mapping the Cognitive Demands of Learning to Program. In D. N. Perkins, J. Lochhead, & J. Bishop (Eds.), Thinking: The Second International Conference. Hillsdale, New Jersey: Erlbaum.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14, 14-29.

Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. Journal of Educational Psychology, 68, 143-150.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. Computing Surveys, 13(11), 121-141.

Nachmias, R., Mioduser, D., & Chen, D. (1986, April). Variables -- An obstacle to children learning programming. Paper presented at the annual meeting of the American Educational Research Association, San Francisco, California.

Papert, S. (1980). Mindstorms: Children, computers, and powerful ideas. New York: Basic Books.

Pea, R. D., & Kurland, D. M. (1984a). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2(2), 137-168.

Pea, R. D., & Kurland, D. M. (1984b). Logo Programming and the development of planning skills (report no. 16). New York: Bank Street College.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-56.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers. Norwood, New Jersey: Ablex.

Perkins, D. N., Martin, F., & Faraday, M. (1986). Loci of difficulty in learning to program (Educational Technology Center technical report). Cambridge, Massachusetts: Educational Technology Center, Harvard Graduate School of Education.

Sleeman, D., Putnam, R., Baxter, J., & Kuspa, L. (1986). Pascal and high school students: a study of errors. Journal of Educational Computing Research, 2(1), 5-23.